P1788 Interval Standard Working Group

What you always wanted to know about decorated intervals

John Pryce

Cardiff University, Wales, UK

Contents

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

Aims of talk

- IEEE-P1788, the forthcoming draft Interval Arithmetic Standard, will use decorations.
- A decoration is data attached to an interval $\boldsymbol{x}$ to report information, not about $\boldsymbol{x}$ as such, but about the process of computing it.
- Decorations are an alternative to global flags as in IEEE-754.
- There has been disagreement on what "about the process of computing it" means, hence the vagueness.
  I aim to describe
  - what P1788 decorations are,
  - how their behaviour may be implemented,
  - examples of their use,
  - disagreements about their definition and design.

Outline

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

What one wants to achieve

- Early on, P1788 chose a "silent" paradigm for function evaluation, namely interval evaluation of a library function partly or wholly outside its domain returns an enclosure of those values that are defined:

$$1/[0,1] \text{ gives } [0,\infty), \qquad \sqrt{[-2,-1]} \text{ gives } \emptyset,$$

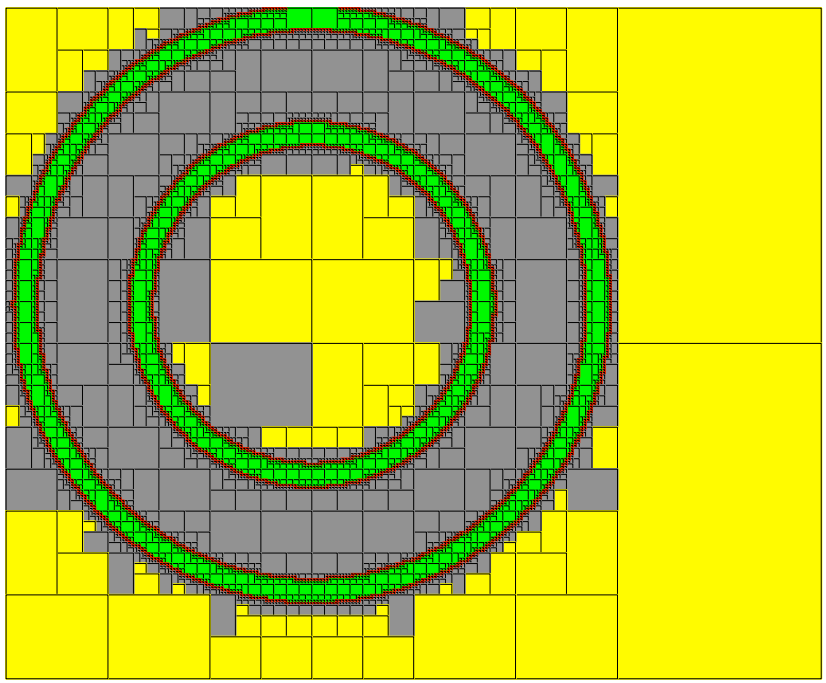
  instead of throwing an exception as in current interval systems.
- Some important interval algorithms require rigorous knowledge about defined-ness and continuity of a function f over a box $\boldsymbol{x}$.
  - Sample app: a graphics rendering algorithm.
    Is f everywhere defined, or by contrast nowhere defined, on $\boldsymbol{x}$?
  - Sample app: validated solution of ODEs.
    Is f is everywhere defined and continuous on $\boldsymbol{x}$?

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

Graphics example

- Given: function $f : \mathbb{R}^n \to \mathbb{R}$, interval $\boldsymbol{y}$, box $\boldsymbol{x}_0 \subset \mathbb{R}^n$. Approximate from inside ("pave") the set $S = \{x \in \boldsymbol{x}_0 \mid f(x) \text{ is defined and } f(x) \in \boldsymbol{y}\}$.
- Done by branch-and-bound. Starting with $\boldsymbol{x} = \boldsymbol{x}_0$ and S empty, evaluate f in interval mode, to do

```
if f is nowhere defined on x
        or f(x) ∩ y = ∅
   discard x
elseif f is everywhere defined
        on x, and f(x) ⊆ y
   insert x in S
elseif x is "small"
   mark x as indeterminate
else
   split x and do the
   same to the pieces
end if
```



(Green: desired set. Yellow: nowhere defined. Gray: $f(\boldsymbol{x}) \cap \boldsymbol{y} = \emptyset$.)

Validated ODEs example

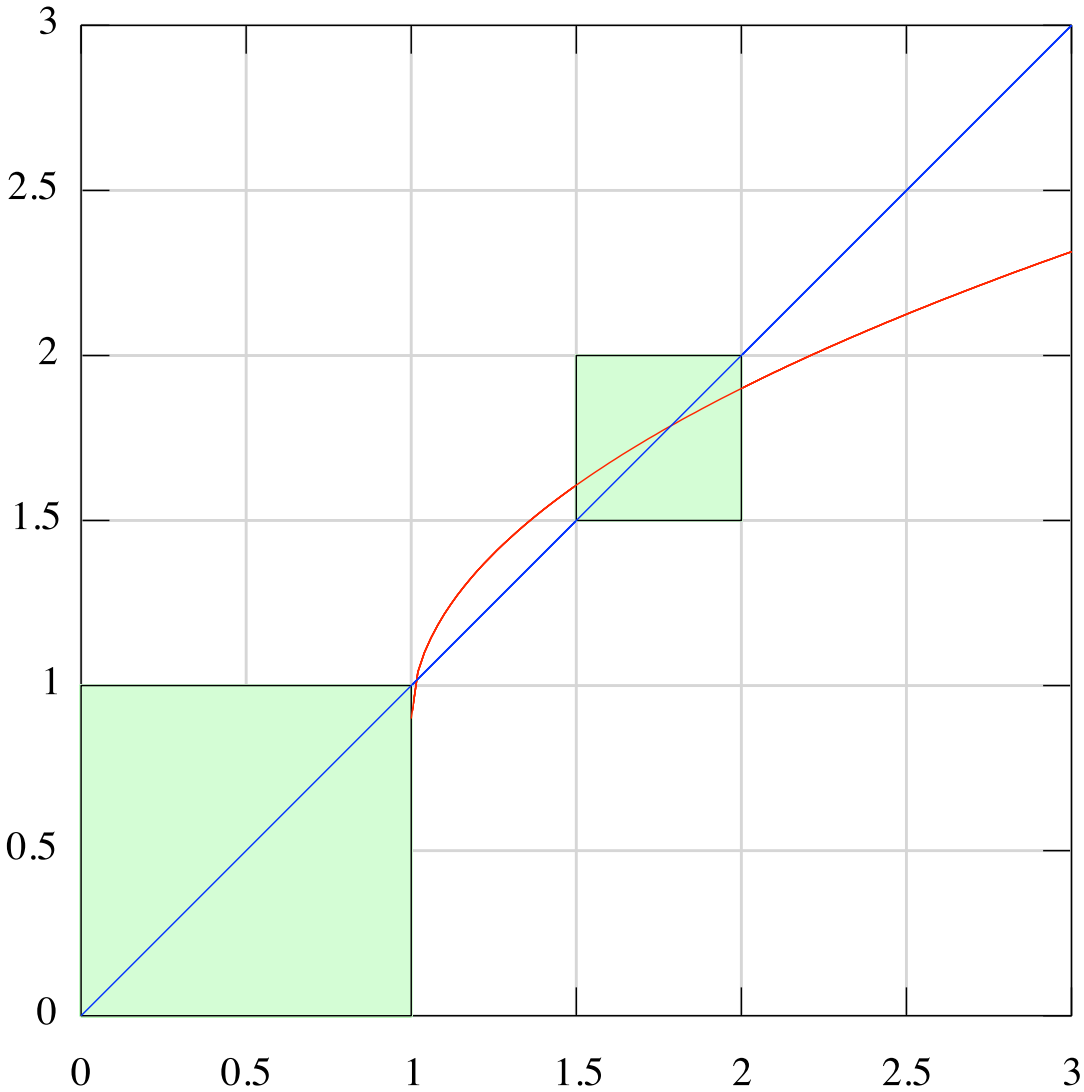
- Vector function $f : \mathbb{R}^n \to \mathbb{R}^n$. Determine if f is everywhere defined and continuous on box $\boldsymbol{x}$.
- If it is, and if interval evaluation gives

$$f(\boldsymbol{x}) \subseteq \boldsymbol{x},$$

one has rigorously verified the conditions of Brouwer's Theorem. So there is a fixed point of f within $\boldsymbol{x}$.

Example

Let $f(x) = \sqrt{x-1} + 0.9$.



Example, cont'd

Apply f to two intervals $\boldsymbol{x}$. Let DAC mean "defined & continuous".

- $\boldsymbol{x} = [1.5, 2]$. Then

$$\boldsymbol{u} = \boldsymbol{x} - 1 = [0.5, 1]; \qquad \boldsymbol{v} = \sqrt{\boldsymbol{u}} \approx [0.7071, 1];$$
$$\boldsymbol{f} = f(\boldsymbol{x}) = \boldsymbol{v} + 0.9 \approx [1.6071, 1.9] \subseteq \boldsymbol{x}.$$

  f is DAC on $\boldsymbol{x}$, so $\boldsymbol{x}$ contains a fixed point.

- $\boldsymbol{x} = [0, 1]$. Then

$$\boldsymbol{u} = \boldsymbol{x} - 1 = [-1, 0]; \qquad \boldsymbol{v} = \sqrt{\boldsymbol{u}} = [0, 0];$$
$$\boldsymbol{f} = f(\boldsymbol{x}) = \boldsymbol{v} + 0.9 = [0.9, 0.9] \subseteq \boldsymbol{x}.$$

  But f is not DAC on $\boldsymbol{x}$, and indeed $\boldsymbol{x}$ doesn't contain a fixed point.

Outline

Getting defined/continuous data automatically

- In interval versions of library functions, include code that determines whether the function is defined and/or continuous on its input interval(s).
- As an expression is evaluated, apply these facts:
  (a) The composition of everywhere defined functions is everywhere defined.
  (b) The composition of everywhere continuous functions is everywhere continuous.
  (c) The composition (in either order) of a nowhere defined function with an arbitrary function is nowhere defined.

  Loosely stated, as no mention of domains & ranges, but gives the idea.
- Item (c) is controversial for “if” expressions, see Appendix.

Revisit the example

$$f(x) = \sqrt{x-1} + 0.9.$$

(FTIA means Fundamental Theorem of Interval Arithmetic.)

- For $\boldsymbol{x} = [1.5, 2]$:
  (a) Subtraction is everywhere DAC, so $v(x) = x - 1$ is DAC on $\boldsymbol{x}$;
  by FTIA its range is $\subseteq \boldsymbol{v} = [0.5, 1]$.
  (b) Sqrt is DAC for $x \geq 0$, in particular on $\boldsymbol{v}$, so
  $w(x) = \sqrt{v(x)} = \sqrt{x-1}$ is DAC on $\boldsymbol{x}$;
  by FTIA its range is $\subseteq \boldsymbol{w} \approx [0.7071, 1]$.
  (c) Addition is everywhere DAC, so $f(x) = w(x) + 0.9$ is DAC on $\boldsymbol{x}$ and
  by FTIA its range is $\subseteq \boldsymbol{f} \approx [1.6071, 1.9] \subseteq \boldsymbol{x}$.

  This can be done automatically using suitably enhanced library functions, and comprises a rigorous proof that

  $$f \text{ is DAC on } \boldsymbol{x} = [1.5, 2] \text{ and maps } \boldsymbol{x} \text{ into itself.}$$

- For $\boldsymbol{x} = [0, 1]$:
  it breaks down at step (b) since sqrt is not DAC on $\boldsymbol{v} = [-1, 0]$.

Where to store the information: local or global?

One can

- Keep one set of global flags to say “continuous”, etc., on lines of IEEE754 flags for Zerodivide, etc.;
  or
- Attach this data to each interval as it’s computed: a local flag, or decoration.

Decorations use more memory & seem cumbersome, but 2 strong advantages are:

- One can compute several functions in parallel—their decorations don’t interfere.
- They handle “dead code” correctly.

Functions in parallel

- An important case is SIMD machines, e.g. compute $f(x) = \sqrt{x-1} + 0.9$ at N (maybe $N = 128$) intervals at once on a GPU or similar hardware.
- Do N (interval) subtracts in parallel, then N sqrts, then N adds. Then inspect N flags and only use results that "succeeded".
- Operating system sees this as one thread of execution so "one flag per thread" doesn't work.
- Decorations handle this in an architecture-independent way.

Dead code example

Same function $f(x) = \sqrt{x-1} + 0.9$.

- Suppose written in Matlab, with a line of dead code added:

```
function y = f(x)
  z = sqrt(-x); % dead, i.e. its output is unused
  y = sqrt(x-1.0) + 0.9;
end
```

  Evaluate with $\boldsymbol{x} = [1.5, 2]$ as before.

- The line that computes ***z*** flags "nowhere defined".
- If stored locally with ***z*** this does no harm.
  If stored in global flag it wrongly says f is nowhere defined on ***x***.

What information to store?

- The original schemes (Hayes, Pryce, . . . 2010) used separate fields to record “the function is everywhere defined”, “the function is nowhere defined”, “the function is everywhere continuous” (over the input box $\boldsymbol{x}$ in each case).
- These were either boolean (a bit), or 3-state logic with a “don’t know” (a trit), or 2 bits joined (4 states, so a tetrit).
  All now dropped.
- We all agreed (de facto) early on that decorations are about properties of a function $f : \mathbb{R}^n \to \mathbb{R}$ and a box $\boldsymbol{x}$ in $\mathbb{R}^n$, jointly.
  I.e. about predicates (= boolean-valued functions)

$$p(f, \boldsymbol{x}).$$

Decorations in a linear order

- Arnold Neumaier devised the idea of organising this in one set of values in a linear propagation order that can be thought of as going from "good" to "bad".
- Simplest useful "scheme" has the 4 decorations

  (good) $\texttt{dac} > \texttt{def} > \texttt{enc} > \texttt{ndf}$ (bad)

  where each denotes a predicate:

Predicate	Short description
$p_{\texttt{dac}}(f, \boldsymbol{x})$	f everywhere defined and continuous on $\boldsymbol{x}$
$p_{\texttt{def}}(f, \boldsymbol{x})$	f everywhere defined on $\boldsymbol{x}$
$p_{\texttt{enc}}(f, \boldsymbol{x})$	always true (short for "enclosing")
$p_{\texttt{ndf}}(f, \boldsymbol{x})$	f nowhere defined on $\boldsymbol{x}$

  (`enc` means "we have enclosure, by FTIA, but that's all we know". Better name welcomed.)
- Equivalently one can think of each decoration as a set,

  $\{\text{all pairs } (f, \boldsymbol{x}) \text{ for which the predicate is true}\}.$

Why this order is useful

- Each stage of evaluating f creates a *conceptual* function, e.g.

  $$u(x) = x, \quad v(x) = x - 1, \quad w(x) = \sqrt{x - 1}, \quad f(x) = \sqrt{x - 1} + 0.9$$

  alongside a *computed* interval

  $$\boldsymbol{u}, \qquad \boldsymbol{v}, \qquad \boldsymbol{w}, \qquad \boldsymbol{f},$$

  that encloses the range of the corresponding function over the input interval—or in general, box—$\boldsymbol{x}$.
- In general at each stage

  $$\text{(new function)} = \phi(\text{some previous functions})$$

  where ϕ is a standard function from the library;
- The point of the order is that decorations of intervals get "worse" as evaluation stages proceed.

Meaning of "decorations get worse"

For simplicity suppose a binary function, so

$$(\text{new function } w(x)) = \phi(u(x), v(x))$$

where u, v are previous functions.

- If we know u and v are `dac` on input box $\boldsymbol{x}$, and ϕ is `dac` on the box in $\mathbb{R}^2$ that is the product of the computed intervals $\boldsymbol{u}$, $\boldsymbol{v}$, then w must be `dac` on $\boldsymbol{x}$.
  Briefly:
  If all of u, v, ϕ are known to be `dac` on their relevant boxes, then w must be `dac` on $\boldsymbol{x}$.
- u, v, ϕ may not be known to be `dac`, but:
  If all of u, v, ϕ are known to be `def` on their relevant boxes, then w must be `def` on $\boldsymbol{x}$ (it might possibly be better).

"decorations get worse" cont'd

- If any of u, v, ϕ is not known to be `def`, and we've "no information" about it, then in general we've "no information" about w.
  I.e., if one of them is `enc` and the others are $\geq$ `enc`, then `enc` is the best we can say about w.
- But there's a state beyond "no information".
  If any of u, v, ϕ is known to be `ndf`, then we know either u or v is nowhere defined on $\boldsymbol{x}$, or ϕ is nowhere defined on the box $(\boldsymbol{u}, \boldsymbol{v})$.
  In all these cases, w is nowhere defined on $\boldsymbol{x}$, i.e., w is `ndf`.
  (Controversial for "if" expressions, see appendix.)

  This condition is terminal—all the partial functions that are "descendants" of w must be nowhere defined on $\boldsymbol{x}$.
- Result: For each standard function ϕ in the expression, we may decorate its output with the *worst*, i.e. smallest in propagation order, of ϕ's *input decorations* and its *local decoration*.

Outline

Implementing such a decoration "scheme"

The key is to enhance interval library functions.

- They must propagate decorations as follows:

  $\boldsymbol{w} = \phi(\boldsymbol{u}, \boldsymbol{v}, \ldots)$ becomes $(\boldsymbol{w}, dw) = \phi\big((\boldsymbol{u}, du), (\boldsymbol{v}, dv), \ldots\big)$

  where dw etc. are decorations and, with "min" in the propagation order,

  $dw = \min(du, dv, \ldots,$ local decoration of ϕ over box $(\boldsymbol{u}, \boldsymbol{v}, \ldots))$.

  This simple rule is sometimes pessimistic, but safe.

- If library functions obey the above rules, [Implementer!]
  and inputs to f are suitably decorated, [User!]
  then Fundamental Theorem of Decorated Interval Arithmetic
  $p_{df}(f, \boldsymbol{x})$ is true,
  where df is the decoration of the output $\boldsymbol{f}$.

Recall $p_{\mathtt{dac}}$ etc are the predicates that define the decorations.

Explain local decoration

- Local decoration is just the continuity & definedness status of ϕ over its input interval(s), derived from ϕ's mathematical definition.
- Examples (intervals $\boldsymbol{u}$, $\boldsymbol{v}$ assumed nonempty, and $b \geq 0$)

Function	Local decoration
$\boldsymbol{u} \pm \boldsymbol{v}$, $\boldsymbol{u} * \boldsymbol{v}$	always `dac`
$\boldsymbol{u}/[0,0]$	`ndf` (/ is undefined on box $(\boldsymbol{u}, [0,0])$ in $\mathbb{R}^2$)
$\boldsymbol{u}/[2,3]$	`dac`
$\boldsymbol{u}/[-1,1]$	`enc`
$\tan([0,b])$	if $b < \frac{\pi}{2}$ then `dac`, else `enc`
$\mathrm{floor}([0,b])$	if $b < 1$ then `dac`, else `def`

- Even in finite precision it's easy to code the local decoration as a byproduct of coding the interval extension, for all common functions.

Initialising the inputs

- The FTDIA may be proved by induction over the number N of operations ϕ in the expression f.
- The base case is $N = 0$, then f is just a variable, say z, defining the real identity function

$$\text{ident}(z) = z.$$

- So to get the base case right we must initialise each input interval, say $\boldsymbol{x}$, with a decoration dx such that

$$p_{dx}(\text{ident}, \boldsymbol{x}) \text{ is true.} \qquad (*)$$

  Then induction works to give a *valid* but maybe pessimistic df.
- For the *best* (most informative) final df, choose the best dx for which (*) holds. For a nonempty $\boldsymbol{x}$, and the "scheme" above, it's

$$dx = \texttt{dac}$$

since ident() is defined and continuous on all of $\mathbb{R}$.
(Empty inputs $\boldsymbol{x}$ have snags, see later.)

Remember widening...

- Interval computing inherently suffers from "widening" due to dependency. This is bound to affect decoration data also.
- So even if f is DAC on $\boldsymbol{x}$ you may not end up with $df = \texttt{dac}$.
  It may be `def`: "all I know is that f is everywhere defined on $\boldsymbol{x}$"
  or even `enc`: "no useful information".
- Put the other way: if you get `def` then `dac` may actually be true.
  If you get `enc`, any of `ndf`, `def` or `dac` may be true.
- In our scheme above, the p_d's are nested within each other:

$$p_{\texttt{dac}} \text{ implies } p_{\texttt{def}} \text{ implies } p_{\texttt{enc}},$$
$$p_{\texttt{ndf}} \text{ implies } p_{\texttt{enc}}.$$

- Current Hayes–WolffvG scheme has decorations logically disjoint:

  their `def` = our (`def` and not `dac`),
  their `enc` = our (`enc` and neither `dac` nor `ndf`).

  But operationally it does exactly what we do so widening has exactly the same potential effects.

Outline

So far, so good, but . . .

- It's not as simple as one would like. Here are some issues:
  - Where does a function begin and end?
  - Empty inputs are a nuisance.

  Of several plausible decoration "schemes", one or two resolve these issues better than the rest.
- Other important points are
  - Disagreement about allowing $\cup, \cap$ in an expression.
  - Impact of decorations on speed and/or storage.

Where does a function begin and end?

Consider a computer program containing interval code.

- In Neumaier–Pryce view a function f is specified by a section of code $\mathcal{F}$ with given start and end points, defining an expression.
  – Initialise input intervals $\boldsymbol{x}_i$ before entering $\mathcal{F}$.
  – Read output decoration df on leaving $\mathcal{F}$.
- Hayes–WolffvG are vague on initialisation: seem happy to initialise decorations "at start of program" and use whatever decorations the $\boldsymbol{x}_i$ happen to have.

Q Maybe they are right??

Function begin/end: empty inputs are a nuisance

- Consider $F(x) = \sin(\mathrm{sqrt}(x))$ evaluated at the input $\boldsymbol{x} = [-2, -1]$.
- $\boldsymbol{x}$ gets initial decoration $dx = \mathtt{dac}$ and we find

$$\begin{array}{rcccccc} (\boldsymbol{v}, dv) & = & \sqrt{(\boldsymbol{x}, dx)} & = & \sqrt{([-2,-1], \mathtt{dac})} & = & (\emptyset, \mathtt{ndf}) \\ (\boldsymbol{F}, dF) & = & \sin((\boldsymbol{v}, dv)) & = & \sin((\emptyset, \mathtt{ndf})) & = & (\emptyset, \mathtt{ndf}) \end{array}$$

- Now suppose the function of interest is really

$$f(v) = \sin(v),$$

  whose input $\boldsymbol{v} = \emptyset$ happens to come from doing $\boldsymbol{v} = \mathrm{sqrt}(\boldsymbol{x})$ "earlier in the program".
- As a decorated interval it is $(\boldsymbol{v}, dv) = (\emptyset, \mathtt{ndf})$.
  Is that a valid initialisation of $\boldsymbol{v} = \emptyset$ for input to f?
- The answer depends on details of the decoration scheme.

Empty inputs, cont'd

- Recall we must initialise $\boldsymbol{x}$ such that $p_{dx}(\text{ident}, \boldsymbol{x})$ is true.
- For the simple scheme above, and $\boldsymbol{x} = \emptyset$, this is ambiguous since *dx* can be either `ndf` or `dac`. (For nonempty $\boldsymbol{x}$, no ambiguity.)
- In a standard, ambiguity must be removed! Here is a tightened up spec of our simple scheme:

Predicate	Short description
$p_{\texttt{dac}}(f, \boldsymbol{x})$	f everywhere defined and continuous on $\boldsymbol{x}$ and $\boldsymbol{x}$ may be $\emptyset$
$p_{\texttt{def}}(f, \boldsymbol{x})$	f everywhere defined on $\boldsymbol{x}$ and $\boldsymbol{x}$ may be $\emptyset$
$p_{\texttt{enc}}(f, \boldsymbol{x})$	always true (short for "enclosing")
$p_{\texttt{ndf}}(f, \boldsymbol{x})$	f nowhere defined on $\boldsymbol{x}$ and $\boldsymbol{x} \neq \emptyset$

- This is how Motion 26 does it, and essentially Motion 27 also.

Empty inputs, cont'd

- But on this scheme $(\emptyset, \texttt{ndf})$, though a possible output of a function, is not a valid input since $p_{\texttt{ndf}}(\text{ident}, \emptyset)$ is false.
  We would have to explicitly re-initialise it for input to $f(v) = \sin(v)$.
- I like a scheme to have the property

  Every possible output is a valid input. (**)

- This is achieved by swapping the "nonempty", "may be empty" conditions in the spec on previous slide:

Predicate	Short description
$p_{\texttt{dac}}(f, \boldsymbol{x})$	f everywhere defined and continuous on $\boldsymbol{x}$ and $\boldsymbol{x} \neq \emptyset$
$p_{\texttt{def}}(f, \boldsymbol{x})$	f everywhere defined on $\boldsymbol{x}$ and $\boldsymbol{x} \neq \emptyset$
$p_{\texttt{enc}}(f, \boldsymbol{x})$	always true (short for "enclosing")
$p_{\texttt{ndf}}(f, \boldsymbol{x})$	f nowhere defined on $\boldsymbol{x}$ and $\boldsymbol{x}$ may be $\emptyset$

- The (unique) initialisation of $\emptyset$ is now $(\emptyset, \texttt{ndf})$.

Empty inputs, cont'd

- This issue exemplifies various small but crucial special cases that take ages to get right.
- The scheme I support has 5 decorations: the four above, plus "ill-formed", `ill`, defined by

Predicate	Short description
$p_{\texttt{ill}}(f, \boldsymbol{x})$	f has empty domain (and $\boldsymbol{x}$ is ignored)

  This has property (**) and, I believe, removes faults of Motion 26 shown by Lefèvre, Hayes etc.
- The main use of `ill` is to express at the mathematical level the result of an invalid constructor call: a "Not an Interval" value.

Property (**) and the domain() function

- Any *nonempty* $\boldsymbol{x}$ can validly be initialised to any of `dac`, `def` or `enc`. Clearly `dac` is best (most informative): e.g. using `enc` on input almost certainly returns `enc` on output, which tells you nothing.
- Neumaier introduced function *domain*($\boldsymbol{x}$) to decorate an interval optimally for use as input to a function f. On the above scheme, it returns $(\emptyset, \texttt{ndf})$ if $\boldsymbol{x}$ is empty, and $(\boldsymbol{x}, \texttt{dac})$ otherwise.
- IMO one reason to like (**) is: it changes *domain*'s status from
  – *essential* to use, for avoiding wrong results, to
  – *sensible* to use, for obtaining optimal results.

- Reminder: Don't think "this is all too complicated". These are 1788 design-time considerations. The user needn't be aware of them.

  George Corliss (paraphrased): "Let us handle the complexity, so the user sees simplicity."

Disagreement about allowing $\cup, \cap$ in an expression

Around 1–2 Aug I had this email exchange with Nate Hayes:

Nate: Also, you have no coherent theory for intersection and union. Saying there should be no standard for these operations is less than a mathematical punt: it is simply not acceptable.

John: I do indeed have a coherent theory, clearly stated. It is that no coherent theory exists. More precisely:
On 1 Jul 2011, I wrote:

> *My rationale is that ALL the various examples so far shown build to the conclusion that union/intersection are not free-standing operations in the way that, say, an interval extension of a point function is. Their decoration logic is too contingent on the environment in which they are used, to be fixed in this standard. If we do fix it, these operations will become illustrations of the proverb*
> *"If all you have is a hammer, everything looks like a nail".*

Disagreement about $\cup$, $\cap$, cont'd

John, cont'd: Please write, and put in your library, a decorated version of intersection that fits your applications. Offer its API for general use. Let others do the same. Just ensure they all have different names. Those that are seen to be useful in several contexts will become popular. At some future date when 1788 is revised, the popular ones can become part of the standard.

As of Sept 2011 this is still my opinion.

Impact of decorations on storage and/or speed

- This is about a feature currently called Bare Object Arithmetic, but could better be Compressed Intervals CI.
- Invented jointly by Hayes & Neumaier, it trades limited decoration features for extra speed.
- A CI value uses same storage as a bare interval, e.g. 16 bytes (=two doubles)
- A decorated interval needs more: 17 bytes? 32 bytes?
  Ian Macintosh has vividly explained the bad effect of “17 bytes” choice in current high-performance architectures, in his recent “subway train” email to P1788 (6 July 2011).

CI arithmetic

- A CI value is either an interval or a decoration—can't hold both at once. (As an IEEE FP value is either a number or a NaN.)
- A CI datum holds an interval until something goes wrong, then "degrades" to a decoration. Controlled by a user-set threshold decoration T:
  - Initially treat intervals as if carrying decoration T.
  - If local decoration of operation is $< T$, then degrade its output.
- Since this can happen in the middle of an expression, CI must handle mixed interval-decoration operations. Clearly (like NaNs)

  $\boldsymbol{x} \bullet \boldsymbol{y}$ must be a decoration if either $\boldsymbol{x}$ or $\boldsymbol{y}$ is a decoration.

Usefulness of CI arithmetic

Compressed Intervals would be a separate data type.
They give sufficient information for the decoration applications sketched earlier, namely

- Brouwer's Theorem to verify existence of fixed point:
  CI takes $T = \texttt{dac}$. It suffices when—as is usually the case—the user is uninterested in the computed box $f(\boldsymbol{x})$ if vector function f is not known to be defined and continuous on $\boldsymbol{x}$.
- Branch & Bound for inner approximation of a set:
  CI takes $T = \texttt{def}$. More problematic.
  Evaluation with full decorated intervals may go to `enc` part-way through an expression on the way to output `ndf`.
  CI wouldn't notice and would output `enc`, thus potentially making B&B do a lot of needless splitting of boxes.
- We are looking at enhancements that remove the B&B disadvantages most of the time.

Status of CI in the Standard

- Typically Compressed Intervals will be 16-byte objects.
- My view: make CI a recommended feature. People who need high-performance intervals will implement it.

Outline

Conclusion

We have done the following.

- Explain why a method of recording defined-ness and continuity information during interval computation is useful, indeed necessary.
- Give sample applications of such recording.
- Argue in favour of local decorations, not global flags.
- Show how it is practical to implement, by enhancing library functions.
- Discuss some problems and sources of disagreement.

Some further issues are in an Appendix.

Outline

The argument about $\cap, \cup$

A common use of intersection is where we look for several ways to write a function as an expression.
Each one gives an enclosure of the range over a box, and we hope that intersecting these will often give a better enclosure than any of them separately.

Expressing the same function in different ways

- If a function f is given by several algebraically "equivalent" expressions, evaluate each of them on box $\boldsymbol{x}$ in interval mode. Since the results, say $\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots$, each enclose range of f over $\boldsymbol{x}$, so does $\boldsymbol{f}_1 \cap \boldsymbol{f}_2 \cap \ldots$.
- Hayes–Neumaier example: $f_1 = f_1(x, y) = \frac{2x+2y}{2x+y}$ can be rearranged to

  $$f_2 = 2 - \frac{2}{2+y/x} \text{ or } \quad \text{or} \quad f_3 = 1 + \frac{1}{1+2x/y}.$$

  Each should give a sharp range enclosure, because it uses the variables x, y only once.
- Snag is that f_2 only defines same function as f_1 where $x \neq 0$, and f_3 only does so where $y \neq 0$.
- So the desired $\boldsymbol{f}_1 \cap \boldsymbol{f}_2 \cap \boldsymbol{f}_3$ formula has various cases to consider.

Basing "decorated intersection" on this case

Neumaier (2010/11/26 position paper) took this as the paradigm, i.e.:

- $\boldsymbol{h} = \boldsymbol{f} \cap \boldsymbol{g}$ is to give maximally useful information when $\boldsymbol{f} = f(\boldsymbol{x})$ and $\boldsymbol{g} = g(\boldsymbol{x})$, and where (point-functions) $f(x)$ and $g(x)$ are mathematically equal at all points x where they are both defined.
- The desirable case is when $\boldsymbol{f}$, $\boldsymbol{g}$ are both decorated `def` or better, since then both are everywhere defined on input box $\boldsymbol{x}$
  ...and their "common range" is $\subseteq$ both $\boldsymbol{f}$ and $\boldsymbol{g}$.
- So it makes sense to return $\boldsymbol{h} = \boldsymbol{f} \cap \boldsymbol{g}$, decorated with the better of the decorations of $\boldsymbol{f}$, $\boldsymbol{g}$, i.e.

$$(\boldsymbol{f}, df) \cap (\boldsymbol{g}, dg) = (\boldsymbol{f} \cap \boldsymbol{g}, \max(df, dg))$$

  (since if either is `dac`, their "common function" is DAC on $\boldsymbol{x}$).
- When either decoration is $<$ `def`, Neumaier used same formula
- ...but when either input is "ill-formed", or the undecorated intersection is empty, a different formula is used.

My critique

- Neumaier gave examples using f_1, f_2, f_3 on Slide 44, showing better enclosures than just using "naive" f_1.
- But each of us found errors in the other's formulation, arising from the Q: What is the "common function"?
  In the example above, should it be defined on the intersection of the domains (i.e., where $x \neq 0, y \neq 0, 2x + y \neq 0$) or their union (i.e., where $2x + y \neq 0$)?
- One scenario, & two experts couldn't agree a simple formulation!
- Another rather different application of intersection is in the interval Newton's method. Here, yet another rule seems appropriate.
- Hence my scepticism about finding a generally useful decorated version of intersection. The case with union is similar.

Controversy on "if" expressions

- A simple way to include conditionals into expressions is to regard boolean values as 0 or 1 in the usual way, and an "if" as an evaluation of the function

  $$\text{case}(b, u, v) = u \text{ if } b \neq 0, \text{ else } v,$$

  (like the function $(b?u : v)$ in C).
- According to standard set theory, both branches get evaluated, which implies f defined by

  $$f(x) := \textbf{if } \text{true } \textbf{then } x \textbf{ else } \sqrt{x}$$

  is undefined when $x < 0$.
  Most people would say it should be the function $f(x) := x$ for all x.
- Should the semantics of expressions be changed in this particular case?